Assignment 3

Report 3

Riccardo Cannistrà	s161155
Oldouz Majidi	s163502
Alessandro Montemurro	s171964
Edoardo Nemni	s171939

Technical University of Denmark
Department of Applied Mathematics and Computer Science
02614 High Performance Computing
January 19, 2018

Contents

1 Statement of Problem

The aim of the assignment is to use GPU for scientific parallel computing. The 2D problems we analyzed are the matrix-matrix multiplication and the Poisson problem using the Jacobi formula. The difference with assignments 1 and 2 is that now, using the GPU, we can obtain a performance which is 8 times higher than the one we could obtain using CPU only. For this purpose, we used CUDA, a high-performance parallel computing platform developed by NVIDIA®. The reason behind the discrepancy between the performances of CPUs and GPUs is that the GPU is specialized for compute-intensive, parallel computation.

Unsng the CUDA vocabulary, the CPU is called *host* and the GPU is said *device*. Another key concept of CUDA is the concept of *kernel*. Kernels are C functions that, when called, are executed many times in parallel by CUDA *threads*.

2 Equipment

2.1 Hardware

We connected to the LSF nodes to run batch jobs and evaluate the performance of the code we developed. This assignment mainly required usage of nodes equipped with GPU devices (2x Tesla V100-PCIE-16GB), therefore we show in tables 1 and 2 their main features (the CPU features didn't change since the previous weeks). As it

Specs	**Value**
CPUs	80 (64 CUDA cores per MP = 5120 total CUDA cores)
Clock Rate	1.38GHz
L2 Cache	6.3K
Total Memory	16GB

Table 1: GPU Specifications

Transfer Type	**Bandwidth**
CPU → GPU	12021.5 MB/s
GPU → CPU	12867.9 MB/s
GPU ↔ GPU	741848.0 MB/s

Table 2: GPU Bandwidth tests with Pinned Memory

can be seen in the referenced tables, the clock rate is much lower than for the CPUs we've been using in the previous assignments, as a proof of a generally lower overall computation power of the GPU. However, it comes with an uncomparable number of multiprocessors (cores), which explains the high parallel computing capability of GPUs that as mentioned in section 1. Further bandwidth tests with pageable memory has been performed and clearly showed much lower values compared to the pinned-memory. This has been exploited by CUDA's Runtime API when allocating memory for host data structures, via the `cudaMallocHost()` function that enables pinned-memory allocation, similarly to what happens by default when allocating memory on the device with `cudaMalloc()`.

2.2 Software

In order to approach to the different aspects of the assignment, different software have been used:

- ThinLinc Client: a software used to remotely connect via SSH to one of the dedicated login nodes;
- Unix bash: the terminal included in the OS and the features it exposes (bash scripting etc.);
- Visual Studio Code and Gedit: code editors used to develop all the needed code;
- NVCC: a compiler used to compile CUDA code (`.cu` files);
- Matlab: a software used to parse raw data file and create visualizations;
- FileZilla Client: an FTP client used to safely transfer files between our Windows computers and the GBar machines.

3 Structure

This section explains the iterative steps performed by the group to approach the problem and organize the needed files.

3.1 Design Choices

- **Makefile**: For the matrix multiplication problem we used the `Makefile` provided by Campusnet and a driver (`matmult_f.nvcc`). We wrote our own `Makefile` for the Poisson problem. The `nvcc` compiler has been used with the highest optimization options;
- **Source Code**: for the matrix multiplication problem, the developed source code represented a library that exposes several functions that can be invoked by the driver; these are: (`matmult_nat`, `matmult_lib`, five versions of `matmult_gpu{x}` and `matmult_gpulib`), and the respective prototypes are included in header files (`.h` files);
- **Batch Scripts**: a set of bash scripts has been created to submit batch jobs to the LSF10 clusters; for the Poisson problem, a `main.cu` file was created, that calls different functions according to which kernel should be executed;
- **Executable information**: the driver for the matrix multiplication library accepts the same arguments as for the first week's assignment: type of the library function (e.g. `gpu1`, `gpu2` etc.) and the matrix sizes `m`, `n` and `k`. The `main.cu` file for the Poisson problem accepts 3 command line parameters: type of kernel

(sequential, naive and multi-GPU versions), the matrix size (for simplicity, assumed to be unique for all matrices A, B and C) and the maximum number of iterations k_{max}

The different outputs collected with several batch script executions allowed us to carry out an intense performance analysis on top of the tuning steps that each function included.

3.2 Analysis and evaluation tools

All the different output files of the batch jobs have been parsed by *Matlab* to display the results in more human-readable line-charts.

4 Discussion

4.1 CUDA Version of Matrix Multiplication

4.1.1 Reference parallelized CPU version of CBLAS

In the first week assignment we had used a sequential version of the matrix multiplication provided by the CBLAS library (`cblas_dgemm()`), used to compare our own implementations with a library version that ideally would guarantee optimal performances. As a first step in the current assignment, it was important to highlight the placement of this version among the basic GPU versions: the first one was a sequential implementation of the matrix multiplication (accomplished by launching the kernel with a single block made up of one thread), whereas the second one was a naive parallel version for which each thread computed a unique element of the resulting matrix C. For sake of reference, we included in the library the CPU version of the native library multiplication, as it was developed for the first week's assignment.

4.1.2 First steps with GPU kernels

As already mentioned, the very first kernel we developed is the `matmult_gpu1`, that sequentially computed the matrix multiplication with a unique thread. This version is practically equivalent to the sequential version computed on the CPU with the native `matmult_nat` function. The comparison experiment we carried out to evaluate how much the computational powers of CPU and GPU are different from each other clearly showed the lower performance of GPUs due to a lower clock rate. A single-threaded GPU kernel has drastically lower performance than a highly optimized CPU parallel version of the algorithm. Figure 1 shows a classic performance evaluation plot, and it's clear that the GPU version looks like almost totally flat compared to the parallelized `cblas_dgemm`. Furthermore, this preliminary evaluation has been carried out with relatively small matrix sizes ($N \geq 500$), since the multiplication performed by the GPU version turned out to be significantly slow.

As a countermeasure, the second GPU kernel already set an important benchmark for GPU parallel computing. The `matmult_gpu2` function computed the multiplication by exploiting a high level of parallelism where each element of the resulting C

gpu2

matrix has been computed by a unique thread. For this kernel and all the ones that follow in this report, a fixed thread block size of 16 has been used, whereas the number of the thread blocks is closely dependent on the block size. We also ensured that input A and B matrices with sizes that are non multiple of the thread block size would still produce the correct output. Although fairly simple, this kernel shows a huge performance improvement when compared to the parallelized CPU version of `cblas_dgemm`, as it can be clearly seen in figure 2. Yet, at a first glance it might seem a nice steadily growing curve, but large improvements are still possible. We tried to figure out how much of the total execution time is spent by transferring data between host and device, which represents a measure of the bandwidth utilization. We ran a profiling experiment with the command `nvprof`, a maximum number of iterations set to 1 and a matrix size $N = 5000$. The results are shown in table 3. As we expected, 77.92% of the total execution time is spent within the kernel, whereas only respectively 15.04% and 7.04% are spent for CPU → GPU and GPU → CPU transfers.

Functions	**Time(%)**	**Time**
mult_kernel_gpu2	77,92%	168.17ms
CPU → GPU	15,04%	32,466ms
GPU → CPU	7,04%	15,183ms

Table 3: NVIDIA Profiler summary for inner execution times of `matmult_gpu2` with $N = 5000$

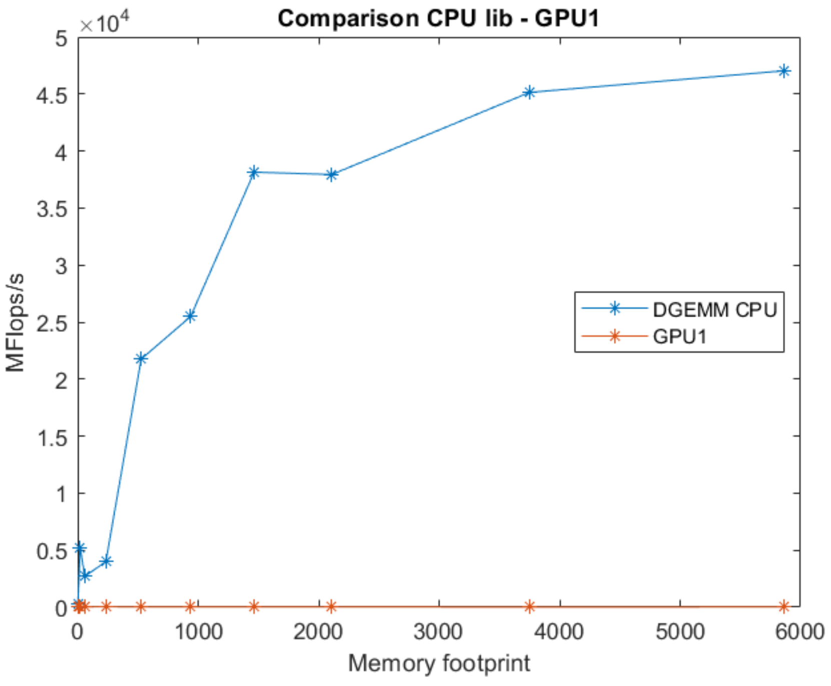

Figure 1: Performance comparison between sequential GPU kernel and parallelized CPU version of `cblas_dgemm`

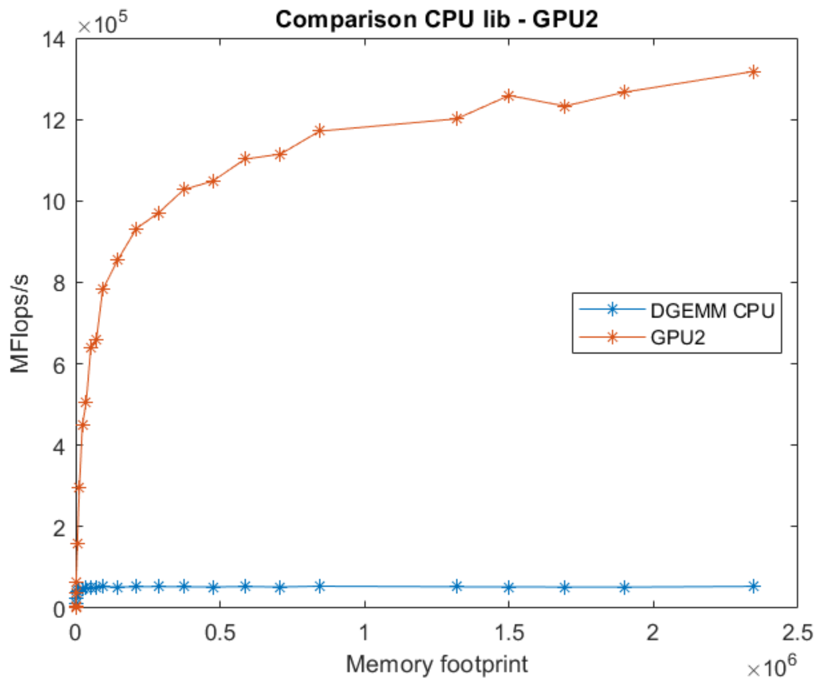

Figure 2: Performance comparison between native parallelized GPU kernel (1 thread per element) and parallelized CPU version of `cblas_dgemm`

4.1.3 Tiled Versions

`gpu2`

A first attempt to improve the `matmult_gpu2` kernel has seen the involvement of tiled matrix multiplication. For this purpose, we first tried to evaluate whether letting each thread compute *exactly* two elements of C would have brought performance improvements. However, achieving the solution was not a straight-forward process. Due to CUDA inner thread block mechanisms, we had to find out which was the optimal way to access matrix elements in a *coalesced* way. The concept of coalesced memory accesses is strictly related to the thread hierarchy: the idea is that a warp of threads should try to perform global memory loads and stores into as few transactions as possible to minimize the bandwidth. The optimal situation would be that neighbor threads within a warp should access sequentially allocated elements in memory. On the other hand, the worst scenario occurs with a strided access. In our case, therefore, we first had to figure out which elements should a single thread compute. Due to time constraints, we only tried with the *bottom* neighbor and the *right* neighbor of an element, and skipped the strided version. Figure 3 shows the results of an experiment carried out to understand which version performed better. On top of these results, we decided to stick with the *bottom* neighbor to compare this first tiled version of matrix multiplication with the previous kernels.

We then tried to extend the 2-tiled version to a more general N-tiled version, where $N > 2$. This implied the choice of an optimal tile size: we indeed experimented with a few values and reached the conclusion that 4 was the best in terms of performances (results in table 4). The general trend is that performances decrease the more the tile size increases. For this reason, the developed `matmult_gpu4` kernel computes matrix multiplication with a tile size of 4, where each thread computes exactly 4 elements that follow the *bottom neighbor* criteria. We didn't have time to experiment with more elements placement. At this stage, we tried to see whether increasing the tile size brought improvements compared to the 2-tiled version and the CPU `dgemm` version. We luckily realized that the code used in `matmult_gpu3` and `matmult_gpu4`, which is fairly similar, was exposing an optimization weakness for loops indexes: originally we preferred to write a more human readable version that was using two loop indexes and represented a drawback for efficient registry utilization. The comparison between `gpu2`, `gpu3`, `gpu4` and the CPU parallelized version of `cblas_dgemm` is shown in figure 4. We didn't obtain results for matrices with size larger than 2000 and this aspect also affected our initial expectations. In the next steps we therefore decided to increase the problem sizes and carry out the experiments on a larger scale, beside fixing the low optimization issue with the loops. At this point, indeed, because of the two reasons just mentioned, the attempt to increase the tile size didn't turn out to bring better results than a tile size of 2. In the next sections it will be more clear how these aspects significantly changed the performance results. It's worth mentioning, however, that both `gpu3` and `gpu4` kernels made use of *registers* to exploit pre-fetching for subsequent data reads.

Tile Size	**MFlops/s (N=32)**	**MFlops/s (N=100)**
3	191,448	5200,110
4	193,839	5337,296
8	186,129	5019,989
16	175,727	4735,498
24	173,327	4519,820

Table 4: Optimal tile size experiment for `matmult_gpu4`

4.1.4 Exploiting Shared Memory

A fifth version, `matmult_gpu5`, has been developed to improve the previous kernels and make use of *shared memory* of thread blocks. The main principle is that shared memory differs from global memory in terms of distance from the thread blocks and the latency that memory reads/writes imply. Since shared memory is *per chip* (thread block), if all threads within a block access data from it this would (ideally) significantly improve the performances. For this reason, `gpu5` includes a blocked version of matrix multiplication, where each thread block has been directly associated to a sub matrix of the result matrix C of size `BLOCK_SIZE` (equal to the size of the thread blocks). Each submatrix C_{sub} is then calculated as the product between two rectangular matrices. These two rectangular matrices are divided into as many square matrices of dimension `BLOCK_SIZE` as necessary and C_{sub} is computed as the sum of the products of these square matrices. These square matrices will be first loaded into shared memory (1 element per thread), and each thread will be in charge of computing one element. A visual representation of this idea can be seen in figure 5. We wrote out own implementation of this kernel, based on the version provided by the official NVIDIA C Programming Guide available online. Again, an experiment has been carried out to evaluate the performance and the results will be finally explained in the next sub-section.

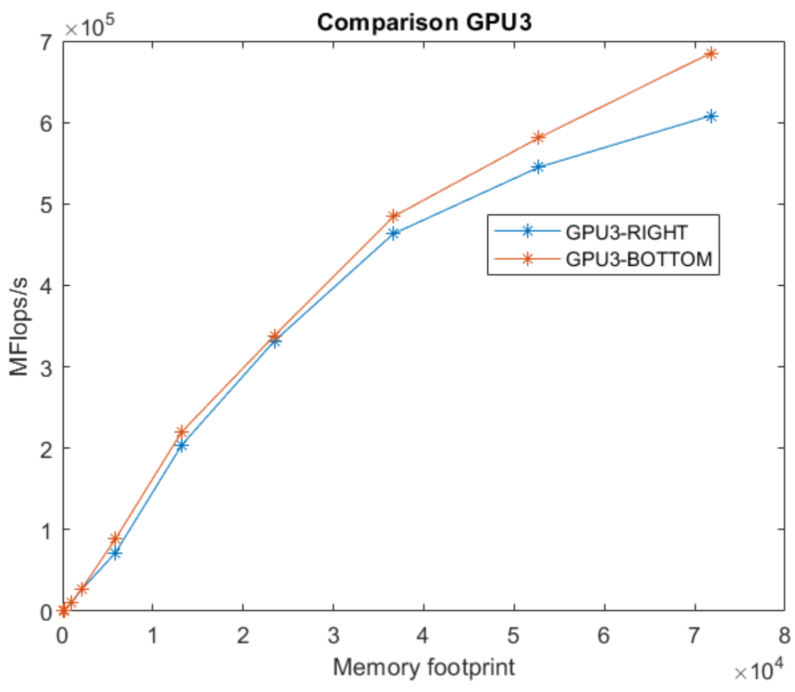

Figure 3: Attempt to find the optimal neighbor that each thread could compute in `matmult_gpu3`

4.1.5 CUBLAS Library `dgemm`

The final step before an overall comparison between all the GPU kernels was developing the `matmult_gpulib` kernel that made use of the GPU library version of CBLAS, called `CUBLAS`, made available by the CUDA Runtime/Legacy API. This version indeed uses the standard `cublas_dgemm` implementation, with all advantages of GPU parallel computation that we've already seen so far.

4.1.6 Overall Evaluation

As it can be seen from figure 6, an interesting and reasonable outcome has been accomplished, after all. By reaching very large matrix sizes (around $N = 10000$, we realized how all the kernels behave and which turns out to be the optimal. As expected, although NVIDIA reported that the `dgemm` CUBLAS version is not yet fully optimized, this turned out to be the best performing kernel. All other kernels follow pretty much the same trend, although it's clear that each of them has a clear placement among the others: `gpu4` performs better than `gpu5`, although the latter makes use of shared memory to minimize latency. `gpu3` and `gpu2` are barely distinguishable, but still perform worse than `gpu5` and `gpu4`.

4.1.7 NVIDIA Profiler

We made use of the *NVIDIA Profiler* tool to analyze our kernels and point out relevant GPU memory issues that significanly affect their performance. Due to time constraints, we can't hereby describe all our findings, however we can point out that a major issue due to low bandwidth utilization seems to be mainly affecting the performances of

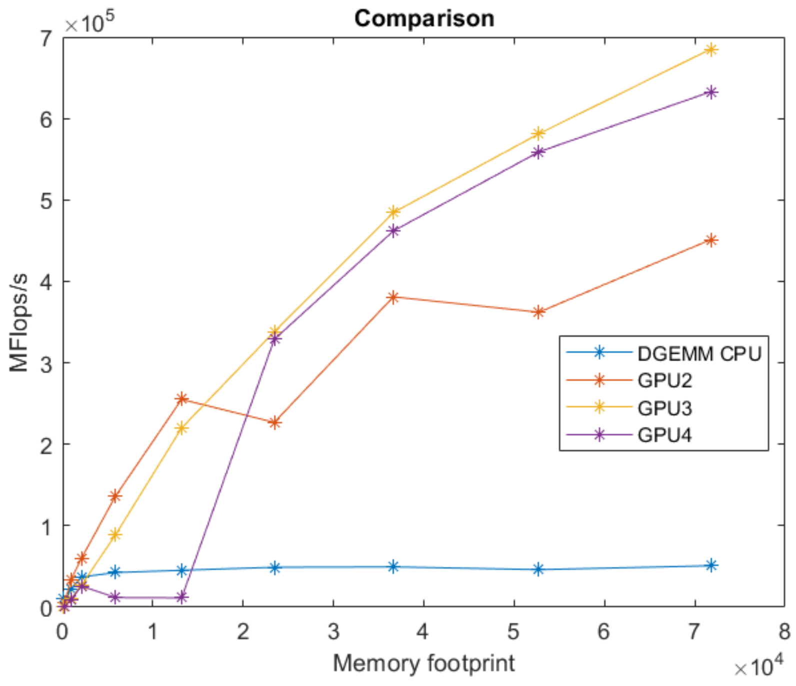

Figure 4: Kernels comparison where `gpu3` and `gpu4` don't take advantage of loop indexes optimizations

pretty much all kernels. This probably relates to the fact that we've been profiling the application with matrix sizes not sufficiently high.

4.2 CUDA Version of Poisson Problem

Table 5 shows a comparison of the performances of the Poisson problem. We reported the execution time in seconds for different problem sizes N. We compared the best version of the CPU version of the problem, using OpenMP, a version using CUDA with only one thread and naive GPU version, i.e. one thread per element. We see that the GPU sequential version has the worst performance; lastly we see that the naive GPU version performs much better than the best CPU version obtained in the first assignment. Lastly, we now want to implement a multi-GPU version of the Poisson problem.

Version	Time ($N = 500$)	Time ($N = 1000$)	Time ($N = 2000$)
CPU OMP Best	0.0625	0.2702	1.2034
Sequential GPU	4.6334	18.4099	339.2083
Naive GPU	0.0012	0.0039	0.0129

Table 5: Comparison of the performances of the Poisson problem between the best CPU version and a GPU version using one thread. The execution time is computed using different sizes of the problem: $N = 500, 1000, 2000$

The function `jacobi_3` provided the needed code host-side: it is used to allocate the memory, initialize the matrices, set the device accordingly, split the matrices in half and assign them to the different devices GPU0 and GPU1. Unfortunately, due to time

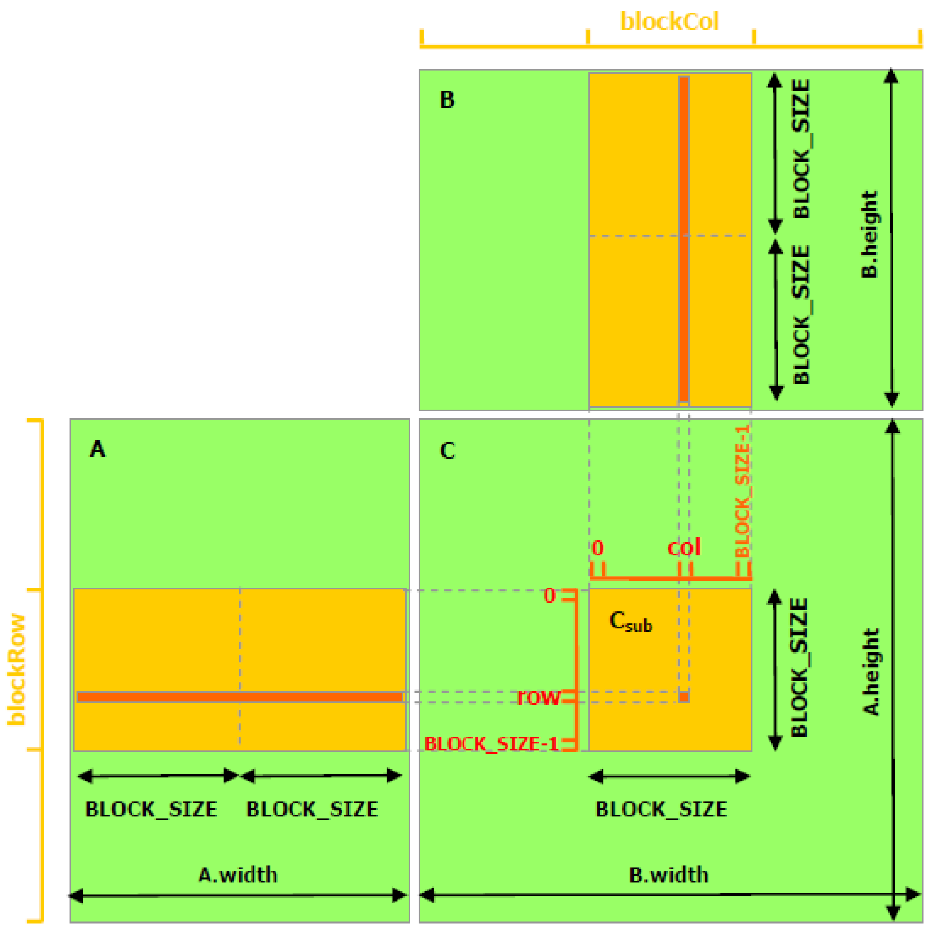

Figure 5: Shared-Memory matrix multiplication in CUDA (from NVIDIA CUDA C Programming Guide

constraints, we couldn't implement the kernel which effectively computes the elements of the matrix.

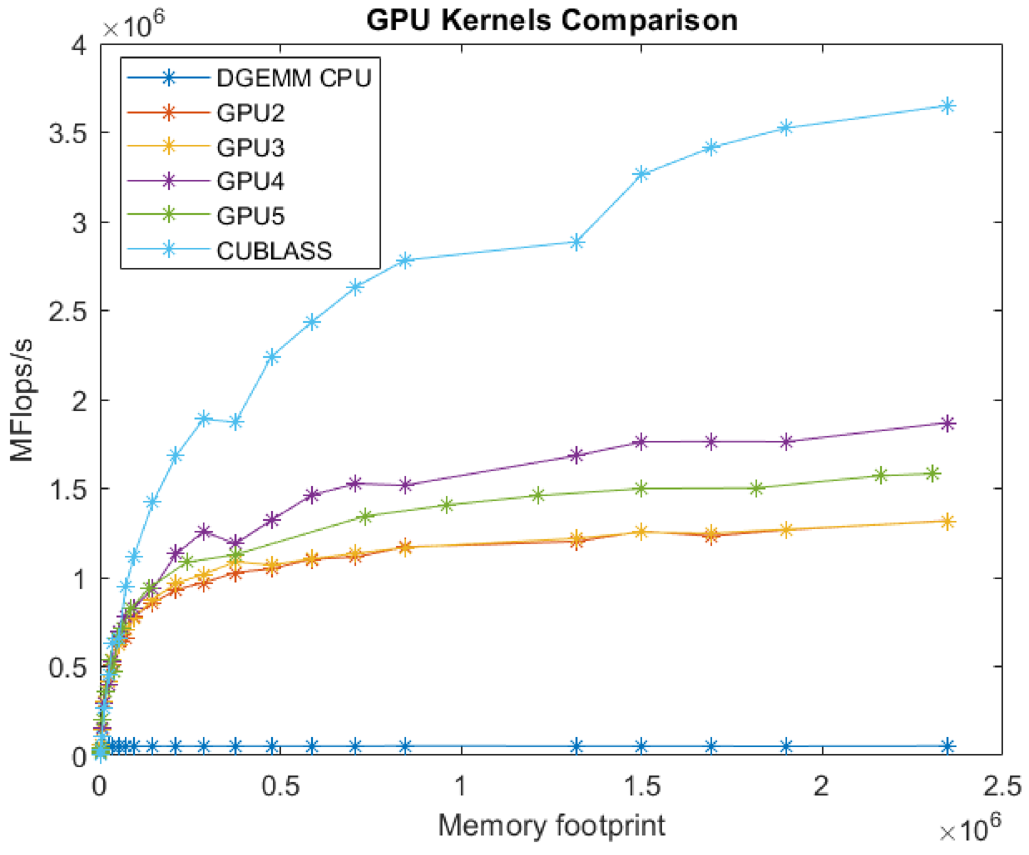

Figure 6: Performance comparison between all relevant GPU kernels